Auto-Vectorization with the Intel Compilers: is Your Code Ready for Sandy Bridge and Knights Corner?

Andrey Vladimirov
Stanford University
for Colfax International

March 12, 2012

Abstract

One of the features of Intel's Sandy Bridge-E processor released this month is the support for the Advanced Vector Extensions (AVX) instruction set. Codes suitable for efficient auto-vectorization by the compiler will be able to take advantage of AVX without any code modification, with only re-compilation.

This paper explains the guidelines for code design suitable for auto-vectorization by the compiler (elimination of vector dependence, implementation of unit-stride data access and proper address alignment) and walks the reader through a practical example of code development with auto-vectorization. The resulting code is compiled and executed on two computer systems: a Westmere CPU-based system with SSE 4.2 support, and a Sandy Bridge-based system with AVX support. The benefit of vectorization is more significant in the AVX version, if the code is designed efficiently. An 'elegant', but inefficient solution is also provided and discussed.

In addition, the paper provides a comparative benchmark of the Sandy Bridge and Westmere systems, based on the discussed algorithm. Implications of auto-vectorization methods for Intel's future Many Integrated Core technology based on the Knights Corner chip are discussed at the end.

Contents

1 Introduction

1.1 Single Instruction Multiple Data (SIMD) Instructions

Most CPU architectures today include Single Instruction Multiple Data (SIMD) parallelism in the form of a vector instruction set. Serial codes (i.e., running with a single thread), as well as instruction-parallel calculations (running with several threads) can take advantage of SIMD instructions and significantly increase the performance of some computations. Each CPU core performs SIMD operations on several numbers (integers, single or double precision floating-point numbers) simultaneously, when these variables are loaded into the processor's vector registers, and a vector instruction is applied to them. SIMD instructions include common arithmetic operations (addition, subtraction, multiplication and division), as well as comparisons, reduction and bit-masked operations (see, e.g., the list of SSE 2 intrinsics). Libraries such as the Intel Math Library provide SIMD implementations of common transcendental functions, and other libraries provide vectorized higher-level operations for linear algebra, signal analysis, statistics, etc.

SIMD instructions can be illustrated with the following *pseudocode*:

Scalar Loop

```
for (i=0; i<n; i++)
  A[i]=A[i]+B[i];
```

SIMD Loop

```
for (i=0; i<n; i+=4)
  A[i:(i+4)]=A[i:(i+4)]+B[i:(i+4)];
```

The SIMD loop above performs $1/4$ the number of iterations of the regular loop, and each addition operator acts on 4 numbers at a time. Such operations can be called by the code via the respective intrinsic function, representing a SIMD addition operator.

One of the most important factors determining the theoretical maximum speedup of a vector instruction set is the width of vector registers. While the first SIMD-capable CPUs in the market featured 64-bit MMX registers, most modern CPUs have 128-bit registers, and newer architectures (AMD's Bulldozer and Intel's Sandy Bridge) have 256-bit AVX registers. The wider the registers, the more numbers of a given type can fit into the register, and thus, the greater the potential speedup. Section 2.4 of this paper shows that the speedup due to vectorization in the latest Sandy Bridge CPU approaches a factor of 8, which is consistent with 256/32=8 single precision floats packed into an AVX vector register.

1.2 Ways to Employ Vectorization

There are several practical ways to incorporate vector instructions into a calculation:

1. Using highly optimized mathematical libraries with vector instruction support. This is the easiest, usually the most efficient, portable way to use the SIMD capabilities of a system. For common mathematical operations, the Intel Developer tools offer libraries such as the Math Kernel Library (Intel MKL) or Intel's Integrated Performance Primitives (Intel IPP). A number of third-party libraries for mathematics employ vector extensions; however, they should be compiled specifically for the architecture (i.e., SIMD instruction set) of the system running the code.
2. When a particular algorithm cannot be represented in the form of standard mathematical operations available in libraries, the programmer can implement vector operations in custom code by
   (a) Explicitly calling SIMD operations functions via intrinsic function calls or inline assembly code,
   (b) Using the Intel C++ Class Libraries to work with data in the form of short vectors, or
   (c) Instructing the compiler to automatically implement vectorization where possible.

Most modern compilers are able to auto-vectorize regular C, C++ or Fortran code. Auto-vectorization has tremendous advantages over explicit calls to SIMD intrinsics. First, it greatly reduces the effort required to write vectorized code. Second, it makes the code more readable. And last, but not least, it makes the code forwards-compatible with future generations of CPUs and SIMD instruction sets.

This paper focuses on auto-vectorization (i.e., case *2(c)*). We will illustrate, using the Intel C++ compiler, that it is easy to facilitate auto-vectorization in suitable calculations. Good performance and forward-compatibility of auto-vectorized code can be achieved by following well-defined vectorization guidelines.

2 Practical Example: Coulomb's Law Calculation

While Coulomb's law strictly applies only to the electric interaction of two charged particles, algorithms computationally similar to Coulomb's law are applied in other fields. For example, the N-body calculation uses the same mathematical form for the gravitational interaction of point masses. And the concept of particles, the combination vector algebra and transcendental functions, and the all-to-all nature of interactions used in this example are staple in mechanical problems, visualization, etc.

The essence of Coulomb's law in the electric potential formulation is as follows. Suppose there are m point charges, carrying electric charges q_i and located at coordinates specified by position vectors $\vec{r}_i$. Then the electric potential, Φ, at a point in space specified by the vector $\vec{R} \equiv (R_x, R_y, R_z)$ is given by the expression

$$\Phi(\vec{R}) = -\sum_{i=1}^{m} \frac{q_i}{|\vec{r}_i - \vec{R}|}, \tag{1}$$

where $|\ |$ denote the magnitude (i.e., length) of a vector:

$$|\vec{r}_i - \vec{R}| = \sqrt{(r_{i,x} - R_x)^2 + (r_{i,y} - R_y)^2 + (r_{i,z} - R_z)^2}. \tag{2}$$

Figure 1 is a visual illustration of the problem. In the left panel, $m = 512$ charges are distributed in a lattice-like pattern. Each of these particles contributes to the electric potential at every point in space. The right panel of the figure shows the electric potential at 128×128 points in the xy plane at $z = 0$, calculated using Coulomb's law.

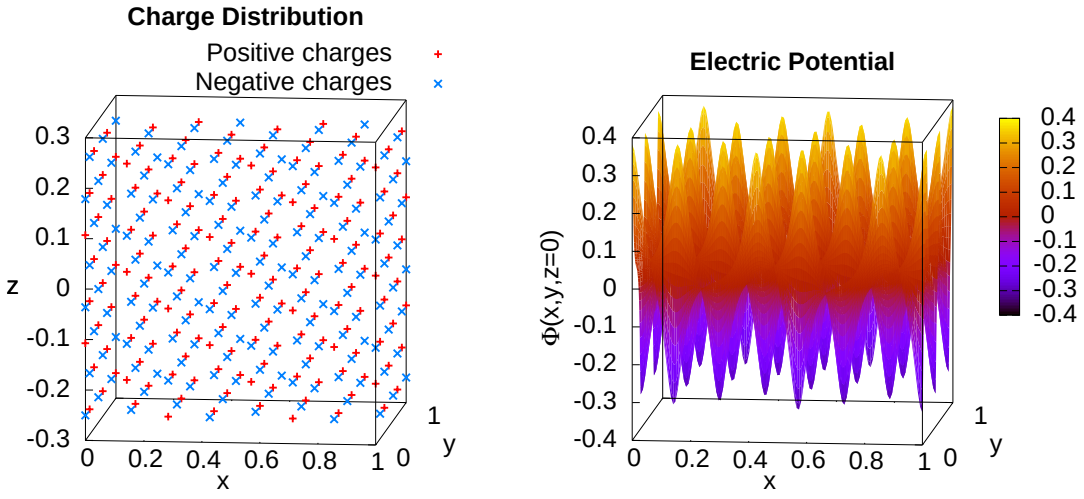

Figure 1: Left panel: a set of charged particles. Right panel: the electric potential Φ in the $z = 0$ plane produced by charged particles shown in the left panel. For every point in the xy-plane, equation (1) was applied to calculate $\Phi(\vec{R})$, where the summation from $i = 1$ to $i = m$ is taken over the m charged particles.

The rest of this section discusses the construction of an efficient C++ code for calculating the electric potential given by equation (1) on a grid.

2.1 Elegant, but Inefficient Solution

As a physicist, I am used to treating particles as the basis of physical models, and therefore I would be tempted to start designing a C++ code by defining the particle class in this way[1]:

```
1  class Charge { // Elegant, but ineffective data layout
2    public:
3      float x, y, z, q; // Coordinates and value of this charge
4      charge();
5      ~charge();
6  };
7  // The following line declares a set of m point charges:
8  charge *Q = new Charge[m];
```

Figure 2

This is an elegant and natural solution; however, as we will see later, it is inefficient when it comes to using this charge distribution to calculate the electric potential using Coulomb's law. The negative impact of this inefficiency is especially noticeable in an AVX-capable CPU. However, in order to demonstrate this, let us proceed by introducing the calculation of the electric potential Φ in the $z = 0$ plane using the array `Q`:

```
1  // This version performs poorly, because data layout of class Charge
2  // does not allow efficient vectorization
3  void calculate_electric_potential(
4      const int m,         // Number of charges
5      const int n,         // Number of points in xy-plane in each dimension
6      const Charge* chg, // Charge distribution (array of classes)
7      float* const phi,  // Output: electric potential
8      const float ds       // Spatial grid spacing
9      ) {
10   for (int c=0; c<n*n; c++) {
11     const float Rx=ds*(float)(c/n); // x-coordinate of observation point
12     const float Ry=ds*(float)(c%n); // y-coordinate
13     const float Rz=ds*(float)(0   ); // observations in z=0 plane
14     for (int i=0; i<m; i++)  { // This loop will be auto-vectorized
15       // Non-unit stride: (&chg[i+1].x - &chg[i].x) != sizeof(float)
16       const float dx=chg[i].x - Rx;
17       const float dy=chg[i].y - Ry;
18       const float dz=chg[i].z - Rz;
19       phi[c] -= chg[i].q / sqrtf(dx*dx+dy*dy+dz*dz); // Coulomb's law
20     }
21   }
22 }
```

Figure 3

When the code is compiled with arguments `-O2` and `-xAVX`, and run on the Sandy Bridge E5-2680 CPU (System 1 in Section 3) with $m = n = 2^{11}$, the calculation takes 11.2 seconds. This is reflected in Figure 6 in the 'Inefficient Vectorization' case for the 'Sandy Bridge: 256-bit AVX' system.

In order to understand why this result can be improved, consider the inner `for`-loop in line 14 of Figure 3. The variable `chg[i].x` in the i-th iteration is `4*sizeof(float)=64` bytes away in memory from `chg[i+1].x` used in the next iteration. This corresponds to a stride of `64/sizeof(float)=4` instead of 1, which will incur a performance hit when the data are loaded into the processor's vector registers. The same goes for members `y`, `z` and `q` of class `Charge`. In addition, we made no effort to ensure proper alignment of data in memory, which may further degrade performance.

[1] Of course, declaring key members of a class as `public` is not a good example of object-oriented programming, and neither is passing class arrays as function arguments in future examples. However, the intent of this paper is to provide guidelines for the facilitation of auto-vectorization, and the methods shown here will be applicable in C++ as well as in C and, to some degree, Fortran 90. To maintain simplicity and compatibility, programming style will sometimes be compromised.

2.2 Optimization: Unit-stride Data Access

In order to achieve unit-stride data access in the inner loop of function `calc_electric_potential`, the structure of data needs to be re-organized. Instead of the inefficient `class Charge`, let us declare a class that contains the properties of charges as arrays:

```
1  class Charge_Distribution {
2    // This data layout permits effective vectorization of Coulomb's law application
3   public:
4    const int m; // Number of charges
5    float * x; // Array of x-coordinates of charges
6    float * y; // ...y-coordinates...
7    float * z; // ...etc.
8    float * q; // These arrays are allocated in the constructor
9
10   Charge_Distribution(const int M);
11   ~Charge_Distribution();
12 };
```

Figure 4

With this new class, the function calculating the electric potential takes on the following form:

```
1  void calc_potential(
2          const int m,     // Number of charges
3          const int n,     // Number of points in xy-plane in each dimension
4          const Charge_Distribution & chg, // Charge distribution (arrays of properties)
5          float* const phi,  // Output: electric potential
6          const float ds  // Spatial grid spacing
7          ) {
8    // This version vectorizes well thanks to unit-stride, aligned data access
9    for (int c=0; c<n*n; c++) {
10     const float Rx=ds*(float)(c/n);
11     const float Ry=ds*(float)(c%n);
12     const float Rz=ds*(float)(0   );
13     for (int i=0; i<m; i++)  {
14       // Unit stride: (&chg.x[i+1] - &chg.x[i]) == sizeof(float)
15       const float dx=chg.x[i] - Rx;
16       const float dy=chg.y[i] - Ry;
17       const float dz=chg.z[i] - Rz;
18       phi[c] -= chg.q[i] / sqrtf(dx*dx+dy*dy+dz*dz);
19     }
20   }
21 }
```

Figure 5

Clearly, the inner `for`-loop in line 13 of Figure 5 has unit-stride data access, as `chg.x[i]` is immediately followed by `chg.x[i+1]` in memory, and the same goes for all other quantities accessed via the array iterator `i`.

The new code successfully compiles, but does it improve performance? In fact, the time to run the function `calc_potential` is now 30.3 seconds. Even though we expected an improvement, this result is worse than in the previous case! This is, of course, a planned failure, staged only to provide motivation for the discussion in the next two subsections.

In fact, Section 2.3 will show that this function performs poorly because vectorization was not implemented by the compiler at all due to a lack of information from the programmer. This case is represented by the 'No Vectorization' group of bars in Figure 6.

However, the optimization made in this section is correct, and only a minor adjustment is necessary to permit auto-vectorization and achieve much better performance. Read on to learn how.

2.3 Optimization: Eliminating Assumed Vector Dependence

In order to understand why the function in Section 2.2 performs worse than the function in Section 2.1 (even though it *should* perform better), let us take a look at the vectorization report. In order to do that, one needs to recompile the code with the compiler argument `-vec-report3`. The output reads (line numbers match those in Figure 5):

```
engine.cc(13): (col. 5) remark: loop was not vectorized: existence of vector dependence.
engine.cc(18): (col. 7) remark: vector dependence: assumed FLOW dependence between phi line 18 and chg line 17.
engine.cc(17): (col. 32) remark: vector dependence: assumed ANTI dependence between chg line 17 and phi line 18.
engine.cc(18): (col. 7) remark: vector dependence: assumed FLOW dependence between phi line 18 and chg line 17.
engine.cc(17): (col. 32) remark: vector dependence: assumed ANTI dependence between chg line 17 and phi line 18.
engine.cc(18): (col. 7) remark: vector dependence: assumed FLOW dependence between phi line 18 and chg line 17.
engine.cc(17): (col. 32) remark: vector dependence: assumed ANTI dependence between chg line 17 and phi line 18.
engine.cc(18): (col. 7) remark: vector dependence: assumed FLOW dependence between phi line 18 and chg line 17.
engine.cc(17): (col. 32) remark: vector dependence: assumed ANTI dependence between chg line 17 and phi line 18.
engine.cc(18): (col. 7) remark: vector dependence: assumed FLOW dependence between phi line 18 and chg line 18.
...
```

Evidently, the compiler has refused to vectorize the inner loop, suspecting that it may contain unvectorizable vector dependences. This suspicion arose because the compiled function was placed in a separate file, and the compiler could not detect whether the pointer `phi` is pointing to one of the members of the variable `chi`, or not. In order to overcome this limitation, the programmer must indicate to the compiler that `phi` is indeed independent from `chi`. There are two ways to achieve that:

1. The `restrict` keyword can be placed in the declaration of the argument `phi` of this function:

   ```
   void calc_potential(
   // ...
           float* restrict const phi,  // Output: electric potential
   //...
   ```

   and the code should be compiled with the compiler argument `-restrict`. This tells the compiler that during the life of that pointer, no other pointer accesses the data referenced by `phi`.

2. Alternatively, `#pragma ivdep`, a hint recognized by the Intel compiler, can be placed before the loop:

   ```
   #pragma ivdep
       for (int i=0; i<m; i++)  {
         //...
       }
   ```

   This pragma instructs the compiler to ignore possible vector dependences in the loop.

Note that in both cases, the programmer promises to the compiler that there is no true vector dependence in the code. That is, that the `i`-th iteration is independent of the result of the `(i-k)`-th iteration ('flow dependence') or `(i+k)`-th iteration ('anti dependence') for `k>0`. If the above methods of dismissing the compiler's assumptions are used to hide a true vector dependence, the code will crash or produce incorrect results.

With the either of the modifications described above (I chose the `restrict` keyword), the code compiles, and the vectorization report reads:

```
engine.cc(13): (col. 5) remark: LOOP WAS VECTORIZED.
```

This is what we want. The execution time is now 4.33 seconds, which is 2.6 times better than for the poorly performing code Section 2.1, and 7.0 times faster than the unvectorized code (Section 2.2). In Figure 6, this case is represented by the group called 'Optimal Vectorization: Unit Stride Access'.

Even though it is a good result, there is still room for improvement with very little programming effort involved. The final optimization for this code is described in Section 2.4.

2.4 Optimization: Data Alignment

Loading data into vector registers is most efficient when the beginning of the data is aligned on a 16- (for 128-bit SSE registers) or 32-byte boundary (for 256-bit AVX registers). Alignment on a 32-byte boundary means that the memory address of the first byte of the array is a multiple of 32. With Intel compilers, aligned arrays can be allocated with the `_mm_malloc` and `_mm_free` intrinsics instead of the `malloc` and `free` calls. In order to properly align the members of the `Charge_Distribution` class, its constructor and destructor must perform the following:

```
Charge_Distribution::Charge_Distribution(const int M) : m(M) {
  x=(float*)_mm_malloc(m*sizeof(float), 32);  y=(float*)_mm_malloc(m*sizeof(float), 32);
  z=(float*)_mm_malloc(m*sizeof(float), 32);  q=(float*)_mm_malloc(m*sizeof(float), 32);
}
Charge_Distribution::~Charge_Distribution(){
  _mm_free(x); _mm_free(y); _mm_free(z); _mm_free(q);
}
```

and a similar aligned allocation must be made for the array `phi`.

An alternative way to achieve alignment is to use the `malloc` call to allocate a block of memory slightly larger than needed, and then point a new pointer to an aligned address within that block:

```
  char *foo=malloc(bytes+32-1); // Not guaranteed to be aligned
  size_t offset=(32-((size_t)(*foo))%32)%32; // From &foo[0] to nearest aligned address
  float *ptr=(float*) ((char*)(foo) + offset); // ptr[0] is aligned on a 32-byte boundary
```

Note that in this case, the pointer `ptr` should be used to access data, but memory must be `free`-d via `foo`.

In C++, the operator `new` does not guarantee alignment. In order to align a C++ class on a boundary, the programmer can allocate an aligned block of memory using one of the methods shown above, and then use the placement version of the operator `new` (note that '`#include <new>`' may be necessary):

```
  void *buf = _mm_malloc(sizeof(myClass), 32); // buf[0] is aligned on a 32-byte boundary
  myClass* ptr = new (buf) myClass; // placing myClass without allocating new memory
```

When alignment of data on the stack is necessary, the `__declspec(align)` qualifier can be used:

```
  __declspec(align(32)) float A[n];
```

allocates a stack array `A` with 32-byte alignment.

In addition to ensuring proper data alignment for vectorization, the programmer can instruct the compiler to disable runtime checks for data alignment. This will reduce the loop set-up time and benefit the performance of auto-vectorized loops. This can be done in one of the following two ways:

1. The statement '`#pragma vector aligned`' can be placed before the vectorized loop:

   ```
   #pragma vector aligned
       for (int i=0; i<m; i++)  {
         //... The compiler will not implement runtime checks for alignment
       }
   ```

   This pragma can be used in combination with the `ivdep` pragma mentioned above.

2. The `__assume_aligned` pragma can be used: the statement

   ```
   __assume_aligned(A, 32);
   ```

   can be placed before the vectorized loop to indicate that A[0] is aligned on a 32-byte boundary.

In the absence of the above hints of alignment, the program will check array alignment at runtime and, if needed, peel off a few iterations in order to run the rest of the loop with fast aligned instructions.

With both modifications described above: data alignment and a hint to the compiler that the data are aligned, the code compiles and runs in 4.01 seconds (group "Optimal Vectorization + Data Alignment" in Figure 6). This is a significant improvement compared to all previous cases: 2.8 times faster than the code without unit stride (Section 2.1), 7.6 times faster than the unvectorized code (Section 2.2) and 1.08 times faster than the code without alignment optimizations (Section 2.3).

2.5 Final Optimized Serial Code

The optimizations outlined in Sections 2.2, 2.3 and 2.4 result in the following code:

```
1  class Charge_Distribution { // class of arrays is more efficient than array of classes
2   public:
3    const int m;       // Number of charges
4    float *x, *y, *z; // Arrays of coordinates of charges
5    float *q;          // Array of charge values
6
7    Charge_Distribution(const int M) : m(M) {
8      x=(float*)_mm_malloc(m*sizeof(float), 32);  y=(float*)_mm_malloc(m*sizeof(float), 32);
9      z=(float*)_mm_malloc(m*sizeof(float), 32);  q=(float*)_mm_malloc(m*sizeof(float), 32);
10   }
11   ~Charge_Distribution(){
12     _mm_free(x); _mm_free(y); _mm_free(z); _mm_free(q);
13   }
14 };
15
16 void calc_potential(
17         const int m,     // Number of charges
18         const int n,     // Number of points in xy-plane in each dimension
19         const Charge_Distribution & chg, // Charge distribution
20         float* restrict const phi,  // Output: electric potential
21         const float ds  // Spatial grid spacing
22         ) {
23   for (int c=0; c<n*n; c++) {  // Unit-stride, aligned data access in loop
24     const float Rx=ds*(float)(c/n);
25     const float Ry=ds*(float)(c%n);
26     const float Rz=ds*(float)(0  );
27 #pragma vector aligned
28     for (int i=0; i<m; i++)  {
29       const float dx=chg.x[i] - Rx;
30       const float dy=chg.y[i] - Ry;
31       const float dz=chg.z[i] - Rz;
32       phi[c] -= chg.q[i] / sqrtf(dx*dx+dy*dy+dz*dz);
33     }
34   }
35 }
36
37 //...
38   float *phi=(float*)_mm_malloc(n*n*sizeof(float), 32);
39   charge_distribution* CD=new charge_distribution(m);
40   init_charge_array(m, *CD); // this function is defined elsewhere
41   timing_start(); // timing functions defined elsewhere
42     calc_potential(m, n, *CD, phi, 1.0f/(float)(n-1)); // Calculation is run here
43   timing_end();
```

This code must be compiled with the compiler argument `-xAVX` to produce auto-vectorized code with AVX instructions, or with `-xSSE4.2` to produce SSE 4.2 code. Auto-vectorization is enabled with the optimization argument `-O2`. In order to enable the `restrict` keyword, the compiler must be given the argument `-restrict`.

Refer to Figure 6 for benchmark results.

2.6 Other Vectorization Considerations

Non-standard Loops

Generally, the only type of loops that the compiler will auto-vectorize is a `for`-loop, with the number of iterations run known at runtime before the start of the loop. Memory access in the loop must have a regular pattern, ideally with unit stride (i.e., contiguous access from iteration to iteration).

Non-standard loops that cannot be auto-vectorized include: loops with irregular memory access pattern, calculations with vector dependence, `while`-loops or `for`-loops in which the number of iterations cannot be determined at the start of the loop, outer loops, loops with complex branches (i.e., `if`-conditions), and anything else that cannot be, or is very difficult to vectorize. Refer to the Programming Guidelines for Vectorization for more information.

Branches in Vectorized Loops

In architectures that have bit-masked vector instructions, loops with simple branches can be vectorized. For example, this loop can be auto-vectorized:

```
1 for (i=0; i<n; i++)
2   if (a[i]>0) b[i]+=c[i];
```

The compiler will produce a code that runs the loop twice. The first run evaluates all branch conditions and creates bit masks, which have the bits for taken branches set to 1, and for not taken branches – to 0. The second run will perform the bit-masked 'add' operation for all elements of `b[i]` and `c[i]`, which will modify only those `b[i]`, for which the respective bit in the mask is set to 1.

Note that sometimes it may be cheaper to leave a loop with branches unvectorized. For example, suppose that the branch is *almost never* taken, and the branches that *are* taken are fairly expensive. The vectorized version will have to perform all the expensive operations, which will be bit-masked away and wasted. On the other hand, a scalar (i.e., non-vectorized) calculation will zoom through the loop, only evaluating branch conditions and occasionally stopping to take branches. Use `#pragma novector` to avoid vectorization in these cases (see below).

Vectorization-Related Compiler Hints and Arguments

The following list contains some compiler pragmas and command line arguments that may be useful for tuning vectorized code performance. The list pertains to Intel's C++ compiler on Linux. Details can be found in the Intel C++ compiler reference[2]. In the PDF version of this article, the keywords below contain hyperlinks to the respective pages of the Intel C++ compiler reference.

- `#pragma simd`
- `#pragma vector always`
- `#pragma vector aligned | unaligned`
- `#pragma vector nontemporal | temporal`
- `#pragma novector`
- `#pragma ivdep`
- `#pragma loop count`
- `_mm_malloc()`, `_mm_free()`
- `__declspec(align)` and `__assume_aligned` keywords
- `restrict` qualifier and `-restrict` command-line argument
- `-vec-report[n]`
- `-O[n]`
- `-x[code]`

[2] http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/2011Update/cpp/lin/index.htm

3 Sandy Bridge vs Westmere: Performance Benchmark

In order to put the performance of Coulomb's law calculation on the Sandy Bridge system in perspective, this section presents benchmarks of various versions of the code on two hardware systems:

System 1 has two Intel Xeon E5-2680 (Sandy Bridge) CPUs at 2.70 GHz. Each CPU has 8 cores, and hyper-threading is enabled.

System 2 has two Intel Xeon X5690 (Westmere) CPUs X5690 at 3.47 GHz. Each CPU has 6 cores, and hyper-threading is enabled.

Both systems were running CentOS 6.2 with the Linux kernel 2.6.32-220.el6.x86_64. The code was compiled using the Intel C++ compiler version 12.1.3 with arguments '`-O2 -restrict -xSSE4.2`' and '`-O2 -restrict -xAVX`' for SSE 4.2 and AVX versions, respectively.

3.1 Code Parallelization

The calculation of the electric potential on a grid using Coulomb's law is an embarrassingly parallel problem. It is, therefore, trivial to distribute the work of the outer loop (line 23 in the listing in Section 2.5) across all available hyper-threads in the system (32 threads for System 1 and 24 for System 2). Note that the data-level parallelism (i.e., vectorization) will be present in each thread. We used the OpenMP library to parallelize the calculation, and only 3 modifications had to be made to the code in order to parallelize it:

1. Including the OpenMP header file (at the beginning of the code):

```
#include <omp.h>
```

2. Adding the following pragma before the outer loop (line 25 in the listing above):

```
#pragma omp parallel for schedule(guided)
```

3. Compiling the code with an additional argument `-openmp`.

In our 2-way NUMA system, an embarrassingly parallel algorithm may benefit from an additional optimization. That is, in order to take advantage of the low latency of CPU-local memory, the affinity of threads and memory accessed by them can be fixed. This, however, is beyond the scope of this study, and even without this optimization, we get adequate parallel scalability.

3.2 Results of Comparison

System 1 (Sandy Bridge) was benchmarked with the SSE 4.2 *and* the AVX version of the code. System 2 (Westmere) was benchmarked with SSE 4.2 version only, because Westmere CPUs do not support AVX. In addition, in order to evaluate the performance of scalar arithmetics, the code was compiled in the form presented in Section 2.2, which failed to vectorize.

The results are shown in Figure 7. The labels above bars show the measured execution time of function `calc_potential`, and the height of the bars is the ratio of the baseline time to the time of a particular benchmark. The timing accuracy was estimated by running the serial code 10 times and the parallel code 40 times. In all cases, the standard deviation of the computation time did not exceed 3%.

The most striking result of Figure 7 is that for the optimized code, the Sandy Bridge system performs 1.6-1.7 times faster than the Westmere system. At the same time, the scalar (non-vectorized) version of the code did not benefit from running on the Sandy Bridge CPU (i.e., speedup$\approx$1.0), and neither did the version compiled with SSE 4.2 vector instructions. It clearly indicates that the Sandy Bridge chip owes its performance edge in single precision to the 256-bit AVX vector registers.

4 Figures

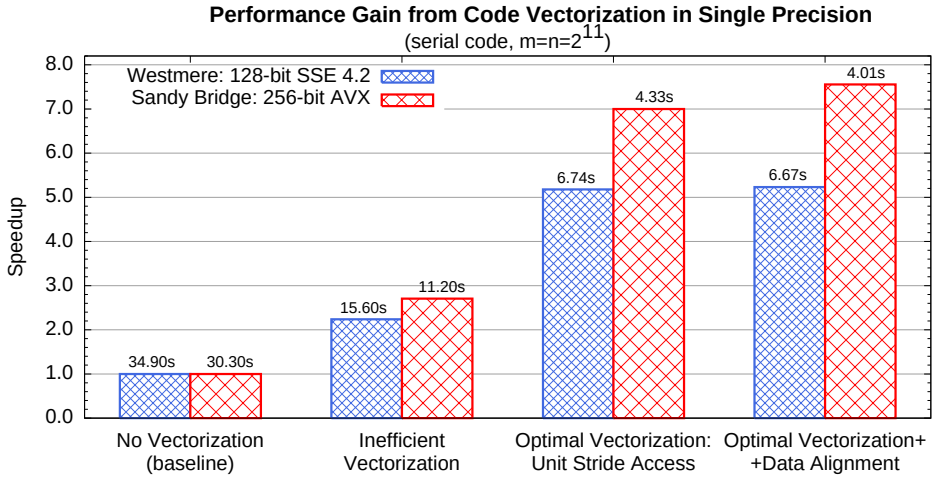

Figure 6: Serial, single precision code was compiled with AVX and SSE 4.2 support and run on Systems 1 and 2 (see Section 3). Labels above bars show the wall clock time of each calculation; bars show the ratio of the baseline time to the time of each case. See text for details: 'No Vectorization' – Section 2.2, 'Inefficient Vectorization' – Section 2.1, 'Optimal Vectorization: Unit Stride Access' – Section 2.3, 'Optimal Vectorization + Data Alignment' – Section 2.4 and 2.5.

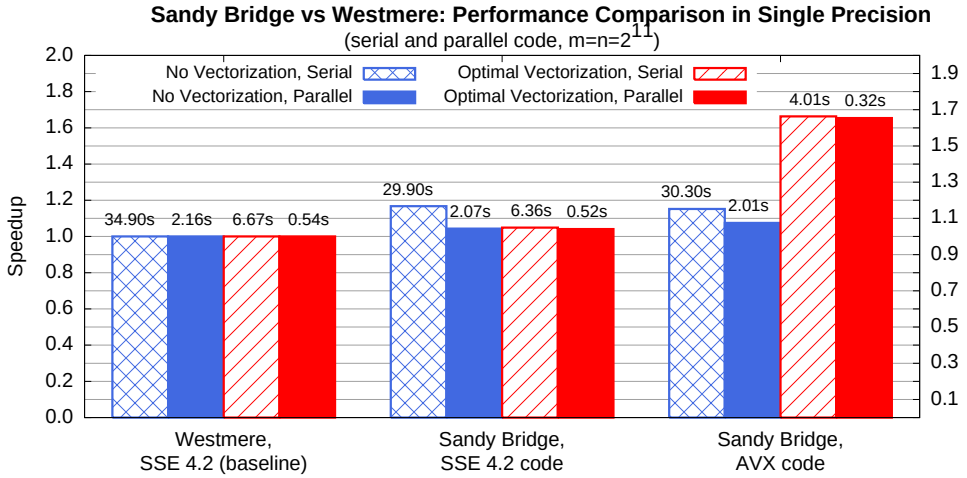

Figure 7: Comparison between System 1 (Sandy Bridge) and System 2 (Westmere) in single precision. See Section 3 for system specifications. Labels above bars show the wall clock time of each calculation; bars show the ratio of the baseline time to the time of each case. See Section 3 for details.

5 Auto-vectorization and the Intel MIC

According to Intel's latest public disclosure about their future product known as Many Integrated Core (MIC) co-processor based on a 22-nm Knights Corner (KNC) chip, "KNC is an optimized, highly parallel co-processor. Unlike traditional accelerators that run portions of programs, Knights Corner is able to run full operating systems and run complex Linux programs". The KNC chip will provide more than 1 TFLOP performance in double precision. Intel has not publicly released any information about the instruction set of the MIC product; however, two statements of Intel's representatives allow one to judge the importance of the approach discussed in this paper.

1. Justin Rattner, Intel's CTO, at Intel Developer Forum (IDF) 2011[3]:

   > ... you'd think you'd have to be some kind of freak to actually program these machines. Or, as we like to say in the labs, you'd have to be a ninja programmer. ... our goal at Intel is really to banish ninja programmers forever.

2. Question and Answer session in the Intel webinar on vectorization with Intel's specialists Shannon Cepeda and Wendy Doerner[4] revealed:

   > **Q**: Intel MIC will have many cores - 50 cores - Will it have such vectorize extensions/ hardware each core? Or the vectorization will be done across different cores?
   > **A**: Your code that uses the high level extensions we covered today on Xeon will scale forward to registers/instructions on the Intel Many Integrated Core (Intel MIC) architecture.

This leaves no doubt that in order to harness the full potential of the MIC product, vectorization is necessary, and that Intel is emphasizing the importance of automated approach to this optimization (i.e., 'high level extensions'), as presented in this paper.

Conclusions

Auto-vectorization of arithmetically intensive loops with compiler directives is a simple and forward-compatible way to greatly improve the performance of a calculation when library functions are not available for the task. The example considered in this paper illustrates how a code can be migrated from the SSE 4.2 instruction set to the next generation architecture, the AVX instruction set, by simply re-compiling it. We have also observed that neglecting to use the new architecture results in a loss of performance, which is a serious argument for using auto-vectorization as opposed to explicit calls to intrinsic functions in CPU codes. Code design that allows for auto-vectorization may also play an important role in the adoption of Intel's Knights Corner-based coprocessors.

Please visit http://research.colfaxinternational.com/ to learn more about the Colfax Research project, comment on this article, and subscribe for updates.

[3] http://download.intel.com/newsroom/kits/idf/2011_fall/pdfs/IDF_2011_Transcript_Justin_Rattner.pdf
[4] http://software.intel.com/en-us/articles/future-proof-your-applications-performance-with-vectorization/